Sorting Algorithms

Aiden, Bryant, Corey, Thomas, Zane

Alternative
Big O Notation

O(1) = O(yeah)
O(logn) = O(nice)
O(n) = O(k)
O(n²) = O(my)
O(2^n) = O(no)
O(n!) = O(mg)
O(n^n) = O(sh*t!)

Approach

- Split up algorithms based on difficulty
- Share the code files through a shared GitHub

Selection Sort

```
void selectionsort(int n, int s[], int &counter)
{
  int temp, i, j, smallest;

  counter = 0;
  for (i = 0; i < n - 1; i++) {
    smallest = i;
    for (j = i + 1; j < n; j++) {
      counter++;

      if (s[j] < s[smallest]) {
        smallest = j;
      }
    }


    temp = s[i];
    s[i] = s[smallest];
    s[smallest] = temp;


    counter++;
  }
}
```

Time Complexity
$O(n^2)$

Average Operations Performed = 50004999

Average Runtime (10,000 element array): 85519μs ≈ 0.085519s

Bubble Sort

```
void bubblesort(int s[], int n, int &counter) {
  int i, j, temp;

  counter = 0;

  for (i = 0; i < n - 1; i++) {
    for (j = 0; j < n - i - 1; j++) {
      counter++;

      if (s[j] > s[j + 1]) {
        temp = s[j];
        s[j] = s[j + 1];
        s[j + 1] = temp;


        counter++;
      }
    }
  }
}
```

Time complexity

$O(n^2)$

Average Operations Performed = 75204154

Average Runtime (10,000 element array): 162260μs ≈ 0.16226s

Heapsort

```
struct heap{
    int size=INT_MIN;
    int* S=nullptr;
    int cnt=0;
    int fcnt=0;

    heap(int newsize){
        size=newsize;
        S=new int[size];
    }

    ~heap(){
        delete[] S;
    }
};
```

```
void heapsort(int n, heap& H){
    makeheap(n,H);
    removekeys(n,H);
    H.fcnt+=2;
}
```

```
void removekeys(int n, heap& H){
    for(int i=n-1; i>=0; i--){
        H.S[i]=root(H);
        H.fcnt++;
    }
}

void makeheap(int n, heap& H){
    H.size=n;
    for(int i=floor(n/2); i>=0; i--){
        siftdown(H,i);
        H.fcnt++;
    }
}
```

```
int root(heap& H){
    int keyout;

    keyout=H.S[0];
    H.S[0]=H.S[H.size-1];
    H.size--;
    siftdown(H,0);
    H.fcnt++;

    return keyout;
}
```

```
void siftdown(heap& H, int i){
    int parent=i;
    int largerchild;
    int siftkey=H.S[i];
    bool spotfound=false;

    while(2*parent+1<H.size && !spotfound){
        H.cnt+2;
        if(2*parent+1<H.size && H.S[2*parent+1]<H.S[2*parent+2]){
            H.cnt+2;
            largerchild=2*parent+2;
        } else largerchild=2*parent+1;

        if(siftkey<H.S[largerchild]){
            H.cnt++;
            H.S[parent]=H.S[largerchild];
            parent=largerchild;
        } else spotfound=true;
    }
    H.cnt++;
    H.S[parent]=siftkey;
}
```

Run-time complexity:

T(n log n)

Unstable sorting:
doesn't preserve relative order

Merge Sort O(n log n)

```
57  void mergeSort(int n, int S[], int &counter)
58  {
59      if (n > 1)
60      {
61          // Find the middle point of the array
62          const int h = n/2;
63          int m = n-h;
64          int U[h];
65          int V[m];
66          for(int i =0; i<h; i++)
67          {
68              U[i] = S[i];
69          }
70          for (int i = h; i < n; i++)
71          {
72              V[i - h] = S[i];
73          }
74          //copy S[1] through S[h] to U[1] through U[h];
75          //copy S[h+1] through S[n] to V[1] through V[m];
76          mergeSort(h, U, counter);
77          mergeSort(m, V, counter);
78
79          // Merge the sorted halves
80          merge(h, m, U, V, S, counter);
81      }
82  }
```

```
2
3   void merge(int h, int m, const int U[], const int V[], int S[], int &counter)
4   {
5       int i, j, k;
6       i=0; j=0; k=0;
7
8       while (i < h && j < m)
9       {
10          //compare
11          counter+=2;
12          if (U[i] < V[j])
13          {
14              S[k] = U[i];
15              i++;
16          }
17          else
18          {
19              S[k] = V[j];
20              j++;
21          }
22          k++;
23      }
24
25      while (i < h)
26      {
27          S[k] = U[i];
28          i++;
29          k++;
30      }
31
32      while (j < m)
33      {
34          S[k] = V[j];
35          j++;
36          k++;
37      }
38  }
39
```

Time to run on different arrays

All done on an array of 10,000 average over 10 runs

1. Randomly distributed elements with little repeating elements
   1,619 microseconds & 241,036 compares
2. Randomly distributed elements with a lot of repeating elements
   1,581 microseconds & 240,798 compares
3. Almost sorted array
   1,010 microseconds & 133,262 compares
4. Reverse Sorted
   959 microseconds & 138,016 compares

Insertion Sort

```
void insertionSort (int n, int s[], int &counter)
{
    int i = 0, j = 0, x = 0;
    for(i = 1; i < n; i++)
    {
        x = s[i];
        j = i - 1;
        while(j >= 0 && s[j] > x)
        {
            s[j+1] = s[j];
            j--;
            counter++;
        }
        s[j + 1] = x;
    }
}
```

Time Complexity: $O(n^2)$ = $(n(n-1))/2$ for average case

Average Operations Performed = 49,725,420 ≈ (10,000(10,000-1))/2 ≈ 49,995,000

Average Runtime (10,000 element array): 52,650.2µs ≈ .05s

Best Case: Array is already sorted
Worst Case: Array is reverse sorted
Average case: Random/Partially sorted

Good for sorting linked lists

Insertion Sort Cases

10,000 Element Array

	sorted	Average (Randomly Sorted)	reverse sorted
operations	10,000 = n	50,418,308 ≈ (n(n-1))/2 = (10,000(10,000-1))/2 = 49,995,000	99,990,000 ≈ n^2 = 100,000,000

Insertion Sort Array Trace

```
9, 9, 7, 6, 5, 4, 3, 2, 1, 0
8, 9, 7, 6, 5, 4, 3, 2, 1, 0
8, 9, 9, 6, 5, 4, 3, 2, 1, 0
8, 8, 9, 6, 5, 4, 3, 2, 1, 0
7, 8, 9, 6, 5, 4, 3, 2, 1, 0
7, 8, 9, 9, 5, 4, 3, 2, 1, 0
7, 8, 8, 9, 5, 4, 3, 2, 1, 0
7, 7, 8, 9, 5, 4, 3, 2, 1, 0
6, 7, 8, 9, 5, 4, 3, 2, 1, 0
6, 7, 8, 9, 9, 4, 3, 2, 1, 0
6, 7, 8, 8, 9, 4, 3, 2, 1, 0
6, 7, 7, 8, 9, 4, 3, 2, 1, 0
6, 6, 7, 8, 9, 4, 3, 2, 1, 0
5, 6, 7, 8, 9, 4, 3, 2, 1, 0
5, 6, 7, 8, 9, 9, 3, 2, 1, 0
5, 6, 7, 8, 8, 9, 3, 2, 1, 0
5, 6, 7, 7, 8, 9, 3, 2, 1, 0
5, 6, 6, 7, 8, 9, 3, 2, 1, 0
5, 5, 6, 7, 8, 9, 3, 2, 1, 0
4, 5, 6, 7, 8, 9, 3, 2, 1, 0
4, 5, 6, 7, 8, 9, 9, 2, 1, 0
4, 5, 6, 7, 8, 8, 9, 2, 1, 0
4, 5, 6, 7, 7, 8, 9, 2, 1, 0
4, 5, 6, 6, 7, 8, 9, 2, 1, 0
4, 5, 5, 6, 7, 8, 9, 2, 1, 0
4, 4, 5, 6, 7, 8, 9, 2, 1, 0
3, 4, 5, 6, 7, 8, 9, 2, 1, 0
```

```
3, 4, 5, 6, 7, 8, 9, 9, 1, 0
3, 4, 5, 6, 7, 8, 8, 9, 1, 0
3, 4, 5, 6, 7, 7, 8, 9, 1, 0
3, 4, 5, 6, 6, 7, 8, 9, 1, 0
3, 4, 5, 5, 6, 7, 8, 9, 1, 0
3, 4, 4, 5, 6, 7, 8, 9, 1, 0
3, 3, 4, 5, 6, 7, 8, 9, 1, 0
2, 3, 4, 5, 6, 7, 8, 9, 1, 0
2, 3, 4, 5, 6, 7, 8, 9, 9, 0
2, 3, 4, 5, 6, 7, 8, 8, 9, 0
2, 3, 4, 5, 6, 7, 7, 8, 9, 0
2, 3, 4, 5, 6, 6, 7, 8, 9, 0
2, 3, 4, 5, 5, 6, 7, 8, 9, 0
2, 3, 4, 4, 5, 6, 7, 8, 9, 0
2, 3, 3, 4, 5, 6, 7, 8, 9, 0
2, 2, 3, 4, 5, 6, 7, 8, 9, 0
1, 2, 3, 4, 5, 6, 7, 8, 9, 0
1, 2, 3, 4, 5, 6, 7, 8, 9, 9
1, 2, 3, 4, 5, 6, 7, 8, 8, 9
1, 2, 3, 4, 5, 6, 7, 7, 8, 9
1, 2, 3, 4, 5, 6, 6, 7, 8, 9
1, 2, 3, 4, 5, 5, 6, 7, 8, 9
1, 2, 3, 4, 4, 5, 6, 7, 8, 9
1, 2, 3, 3, 4, 5, 6, 7, 8, 9
1, 2, 2, 3, 4, 5, 6, 7, 8, 9
1, 1, 2, 3, 4, 5, 6, 7, 8, 9
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Quicksort

```cpp
// quicksort.h

void part(int low, int high, int& pivot, int list[], int& counter){

    int target = low; //Target, which loc in the list where the pivotitem swap to at the end of part
    int pivotitem = list[low]; //Pivotitem, the item to get swapped into its final position after part

    for(int i = low + 1; i <= high; i++){
        if(list[i] < pivotitem){
            target++;
            int temp = list[target]; //Increment the target and swap list[target] and list[i]
            list[target] = list[i];
            list[i] = temp;
            counter++;
        }
    }

    pivot = target;
    list[low] = list[target]; //Swap list[target] and list[low]
    list[target] = pivotitem;
    counter++;

}

void quicksort(int low, int high, int list[], int& counter){ //Calling quicksort with low, high, and the list[]

    int pivot;
    if(low < high){
        part(low, high, pivot, list, counter);
        quicksort(low, pivot - 1, list, counter);
        quicksort(pivot + 1, high, list, counter);
    }

}
```

Time Complexity

O(N ln(N))

Quicksort Cases

Average time complexity for different 10 sets of 10,000 elements

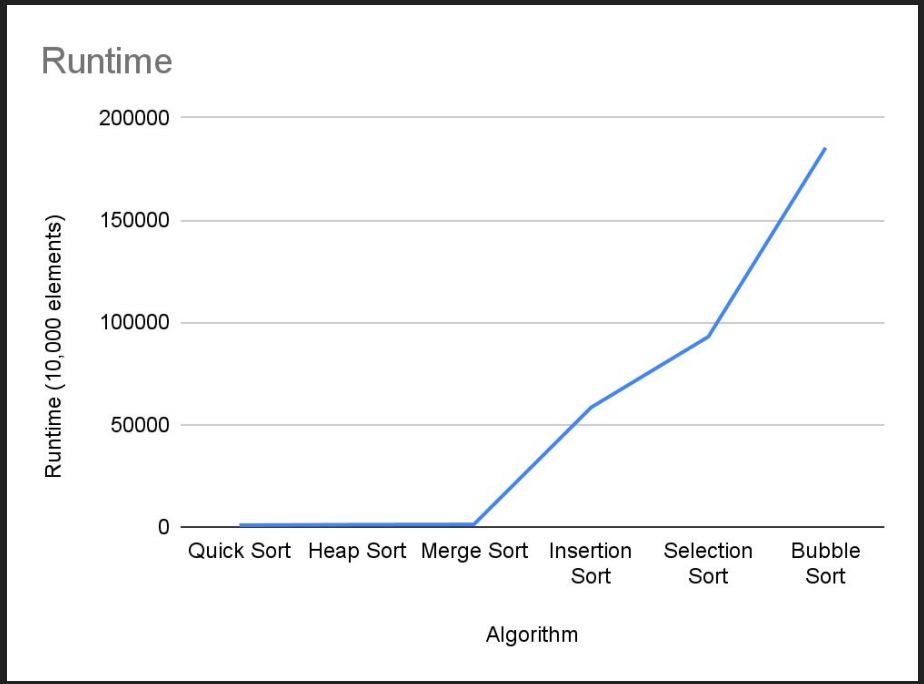
	Average Time in Microseconds	Average Number of Comparisons
Almost Sorted Array	119,981 Microseconds	10,001 Operations
Randomly Distributed Array	1,302 Microseconds	83,942 Operations
Reverse Sorted Array	177,032 Microseconds	25,009,999 Operations
Array with Duplicates	8,581 Microseconds	33,583 Operations

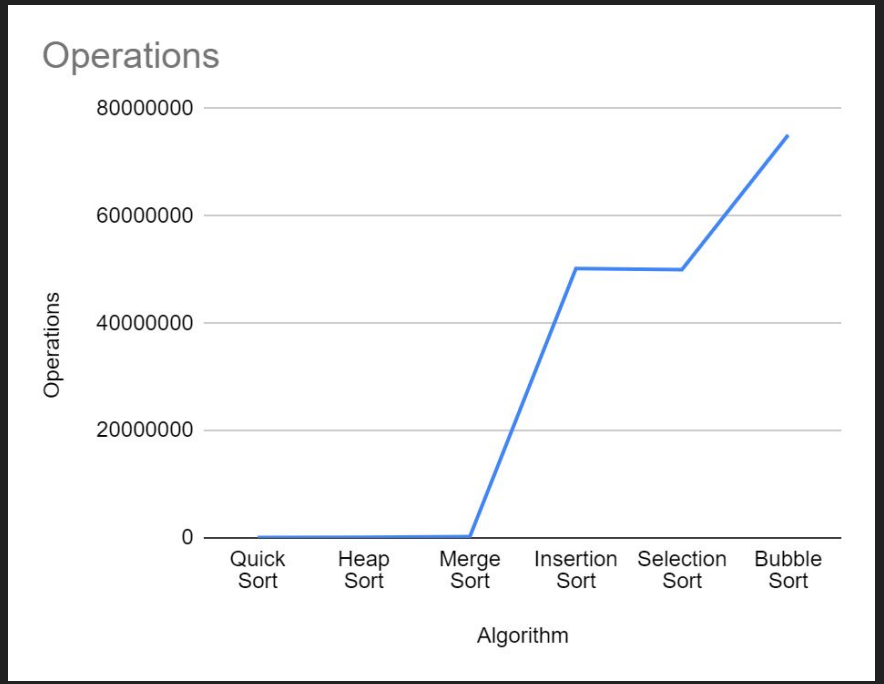
Summary of runtime & Operations

Output of Main (10,000 Randomly Sorted Element Array)

```
Heapsort time taken: 1146 microseconds
Heapsort had: 129275 compares

Quicksort runtime: 894 microseconds
Quicksort performed 82687 operations.

Merge sort runtime: 1222 microseconds
Merge sort performed 240854 compares.

Bubble runtime: 160976 microseconds
Bubble performed 74915979 operations.

Selection sort runtime: 85973 microseconds
Selection sort performed 50004999 operations.

Insertion sort runtime: 49052 microseconds
Insertion sort performed 99683916 operations.
```

Runtime and Operations





Conclusion

Success! All algorithms were properly implemented, and worked as intended.

The results lined up well with what was expected in regards to big O estimations.